

Express Mail<sup>®</sup> mailing label No. EL 521 509 296 US  
Date of Deposit December 12, 1998  
I hereby certify that this paper or article is  
being deposited with the United States Postal  
Service "Express Mail Post Office to Addressee:  
service under 37 CFR 1.10 on the date indicated  
above and is addressed to the assistant  
Commissioner for Patent Washington, D.C. 20231.

Srinivas Matangi  
Typed or printed name of person mailing paper or  
article

[Signature]  
Signature of person mailing paper or article

-1-

PATENT  
Atty Dkt. No. 033144-004

## TD/TDX UNIVERSAL DATA PRESENTATION SYSTEM AND METHOD

### BACKGROUND OF THE INVENTION

#### 1. Field of the Invention

5 The present invention generally relates to data presentation within a  
computing environment and, more particularly, to a platform independent,  
hardware architecture independent and language independent data container which  
provides a wide range of access methods to manipulate and aggregate structured  
and unstructured data.

#### 2. Description of the Related Art

10 A document object model is a programming interface specification which  
allows the programmer to create and modify HTML pages and XML documents as  
a program object containing the contents and data within the object. Document  
objects are data containers used in the exchange of data between computing  
environments.

15 Currently there are two approaches in the presentation of a document object  
model. The first approach is a very general language independent method that  
manipulates the document object by using an ASCII text presentation, such as  
HTML and XML. The second is a language specific implementation that is  
narrowly tailored to particular language models.

20 In order for data to be transmitted and used by different computing  
systems, the generic first approach converts the data into text, such as an ASCII  
state, for example. One disadvantage of this general text presentation is that a  
tremendous amount of memory is used because the data is presented in the form of

strings. Another disadvantage is that the processing time required is substantial longer thus resulting in much slower computing times. The decreased speed takes place because of the time required to convert the represented data into data which can be used by internal computing processes. By way of example, when dealing  
5 with an integer, the data is transformed from an integer to a string representation in order to be transmitted and then back from a string to the integer after transmission. Thus at least two data presentation transformations are needed with every process which uses large quantities of memory and creates unnecessary time delays.

10 The second approach is confined to a specific presentation language, such as JDOM for example. The disadvantage of being language specific is that the implementation of the document object model is limited and dependent upon the specific language. For instance, JDOM is limited to a very specific implementation of a J document, the document model for JAVA and could not be  
15 used in any other type of environment. As such, every language must have a specific document object model associated with it.

Therefore there remains a need to provide for a document object model that may be used universally among languages while maintaining speed and efficient memory utilization.

20

### SUMMARY OF THE INVENTION

The present invention overcomes the shortcomings of the prior art by providing a method for presenting data within a computing environment including an application program interface. The method includes creating a tagged data  
25 object for storing data, encapsulating a data element into the tagged data object to provide a tagged data, and packing the tagged data by converting the tagged data into a binary representation of the tagged data. The tagged data includes a corresponding tag id and the encapsulated data element.

Another aspect of the present invention provides a method for presenting data within a computing environment including an application program interface. The method includes unpacking a packed tagged data by converting the packed tagged data from a binary representation into a tagged data, creating a tagged data object for storing the tagged data, and extracting a data element from the tagged data.

Yet another embodiment provides a method for presenting data within a computing environment of the type having an application program interface prescribed by a data conversion and a wire formatting specification. The method includes creating a tagged data object, encapsulating a data element into the tagged data object to provide a tagged data, packing the tagged data into a binary representation to provide a tagged data transmission, transmitting the tagged data transmission, unpacking the tagged data transmission from the binary representation to retrieve the tagged data, and extracting the data element from the tagged data. The tagged data object may be a universal data container that is platform independent, hardware architecture independent and language independent. The tagged data object may provide broad access to the manipulation and aggregation of data, for example structured data and unstructured data. Upon encapsulation, the tagged data object includes a corresponding tag identifier and the data element.

For a better understanding of the present invention, together with other and further objects thereof, reference is made to the following description, taken in conjunction with the accompanying drawings, and its scope will be defined in the appending claims.

### BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a flowchart describing a method for presenting data within a computing environment including an application program interface;

Fig. 2 is a flowchart describing a method for packing a simple object;

5 Fig. 3 is a flowchart describing a method for packing a complex object;

Fig. 4 is a flowchart describing a method for packing a list object;

Fig. 5 is a flowchart describing a method for encapsulating a data element;

Fig. 6 is a flowchart describing a method for presenting data within a computing environment including an application program interface;

10 Fig. 7 is a flowchart describing a method for unpacking a simple object;

Fig. 8 is a flowchart describing a method for unpacking a complex object;

Fig. 9 is a flowchart describing a method for unpacking a list object;

Fig. 10 is a flowchart describing a method for extracting a data element form the tagged data; and

15 Fig. 11 is a flowchart describing a method for presenting data within a computing environment of the type having an application program interface prescribed by a data conversion and a wire formatting specification.

### DETAILED DESCRIPTION OF THE INVENTION

20 In accordance with the presently claimed invention, a method is provided which combines the generic approach of the general HTML/XML method with a more efficient and faster speed than that of an environment specific presentation. Fig. 1 is a flowchart describing a method for presenting data within a computing environment including an application program interface and is generally designated by the numeral 10. The method 10 begins by creating a tagged data object, as  
25 indicated at 12. The first step is taking pieces of data and converting those pieces of data into a TD internal data representation. The tagged data object may optionally be a composite of three types: a shell object, a complex object and a list object. The shell object may store primitive data types. An example of some of

the types of primitive data may be as follows: an integer, a float, a byte value, a character sequence, a null value, binary data, a string, XML text, a java object, a Td Object, a C/C++ data object, and a data object. The shell object may be a data wrap around. The complex object may be a named tree storing elements under a field name which may be connected to a value. The complex object value may include a shell object. The list object may be a combination between the shell object and the complex object. Thus, when the tagged data object is created, the type of object to be created is determined based upon the type of data to be encapsulated.

After recognizing the type of data, said data is then encapsulated into the data object, as indicated at 14. Optionally, the number of data sequences to be tagged may be determined before converting a particular value to the tagged data structure. The tagged data is labeled data that is physically written into a specific form. By way of example, encapsulating data into a tagged data simple object could be defined as follows: `TdObject x = new TdObject (500)`, where 500 is an integer value.

Another example would be encapsulating data into a tagged complex object for a structure called "Employee" with field names "Last Name", "First Name", and "Salary". As such, the tagged data complex object may be defined as follows:

```
TdObject emp = new TdObject ("Employee");  
emp.add Field ("Last Name", "Smith");  
emp.add Field ("First Name", "Mike");  
emp.add Field ("Salary", 60,000);
```

By way of example, a list object may be defined as follows: `TdObject y = new TdObject ( ); y.add Object (emp); y.add Object (x)`.

The sequence of data is tagged which enables the computer to recognize exactly what type of data is being represented and how many bytes the computer should read after the tag, depending on the type of data. Next, method 10 packs the tagged data into a binary representation, as indicated at 16. The binary

representation may be a wire format which enables the object to be transferred from one computing environment to another. Instead of having to convert the data into a string representation such as ASCII in an HTML/XML approach, the instant invention converts the tagged data into a binary representation of the tagged data object when wire formatting. Optionally, this may be done by identifying the byte as follows: `byte[ ] data = emp.pack ( )` which may return back an array.

Fig. 2 is a flowchart describing a method for packing a simple object and is generally designated by the numeral 20. Method 20 begins with retrieving a simple object source identifier length, as indicated at 22. The simple object size is retrieved, as indicated at 24. Next the simple object type is read, as indicated at 26. Then the simple object value is acquired, as indicated at 28. A section of packed memory is then allocated to accommodate the length of the simple object by using the simple object source identifier, as indicated at 30. The simple object size, type and value are then copied to the packed memory location, as indicated at 32. Once the simple head value and the simple exit value of the packed memory location are acquired, as indicated at 34, those values are then written into the packed memory location, as indicated at 36.

Fig. 3 is a flowchart describing a method for packing a complex object and is generally designated by the numeral 40. Method 40 begins with retrieving a complex object source identifier length, as indicated at 42. The complex object field type is retrieved, as indicated at 44. Next the complex object field value is read, as indicated at 46. This field value may be a simple object. Then the complex object field value is acquired, as indicated at 48. A section of packed memory is then allocated to accommodate the length of the complex object by using the complex object source identifier, as indicated at 50. The complex object field type, size and value are then copied to the packed memory location, as indicated at 52. Once the complex head value and the complex exit value of the packed memory location are acquired, as indicated at 54, those values are then written into the packed memory location, as indicated at 56.

Fig. 4 is a flowchart describing a method for packing a list object and is generally designated by the numeral 60. Method 60 begins with retrieving a list object source identifier length, as indicated at 62. Next a section of packed memory is allocated to accommodate the length of the list object by using the list object source identifier, as indicated at 64. A list object array is then retrieved, as indicated at 66. The list object array is copied into the packed memory location, as indicated at 68. Next the list head value and the list exit value of the packed memory location are acquired, as indicated at 70, and those values are then written into the packed memory location, as indicated at 72.

Fig. 5 is a flowchart describing a method for encapsulating a data element and is generally designated by the numeral 80. Method 80 begins by determining the type of data to be encapsulated, as indicated at 82. The type of data may include an integer, a float numeric value, a one byte value, a character string, a zero terminated character sequence, a byte sequence, a binary data, a null value, a java object, a TD object, an XML text object, a simple (primitive) data type, a compound data type, and a list data type having a combination of data types. Once the data is characterized, the data is then associated with a corresponding tag identifier, as indicated at 84. By way of example, the tag identifiers may include TD\_short, TD\_ushort, TD\_long, TD\_ulong, TD\_float, TD\_double, TD\_byte, TD\_cstring, TD\_blob, TD\_null, TD\_llong, TD\_longstr, TD\_java\_object, TD\_object and TD\_xmlstr.

Optionally, the following definitions may correspond to the above mentioned examples of tag identifiers:

TD\_SHORT: identifies signed short integer that occupies 2 bytes;

TD\_USHORT: identifies unsigned short integer that occupies 2 bytes;

TD\_LONG: identifies signed long integer that occupies 4 bytes;

TD\_ULONG: identifies unsigned long integer that occupies 4 bytes;

TD\_FLOAT: identifies signed float numeric value (with decimal point) that occupies 4 bytes;

TD\_DOUBLE: identifies signed float numeric value (with decimal point) of double precision that occupies 8 bytes;

TD\_BYTE: identifies any one byte value;

5 TD\_CSTRING: identifies any zero terminated character sequence to support compatibility with C/C++;

TD\_BLOB: identifies any byte sequence / binary data;

TD\_NULL: identifies null value (no value);

TD\_LLONG: identifies signed long integer of double precision that occupies 8 bytes;

10 TD\_LONGSTR: identifies long zero terminated character sequence longer than 256 bytes to provide compatibility with long strings in a database;

TD\_JAVA\_OBJECT: identifies any java object;

TD\_OBJECT: identifies any TdObject; and

15 TD\_XMLSTR: identifies any character sequence than may be interpreted as XML text.

Next the tag identifier and the data element are written into the tagged object, as indicated at 86. For example, this can be done by add(data,position,TD). By way of example, writing tagged data for an integer includes an integer tag and an integer value. The tagged data representation may  
20 be represented by <<int>,i> where the "int" is the integer tag and the "i" is the integer value. Another example of tagged data would be a float tagged data which includes a float tag "float" and a float value "f" and may be represented as <<float>,f>. In another example, a string tagged data may be represented as <<string,l>s> where "string" is the string tag, "l" is the string length and "s"  
25 is the string value.

Fig. 6 is a flowchart describing a method for presenting data within a computing environment including an application program interface and is generally designated by the numeral 90. Method 90 begins with unpacking a packed tagged data; as indicated at 92. Next a tagged data object is created for the storage of the



tagged data once the data is retrieved, as indicated at 94. The tagged data object created corresponds to the type of packed tagged data. By way of example, the type of packed data may include a simple type, a complex type, and a list type. Finally, the data element is extracted from the tagged data and placed into the tagged data object, as indicated at 96.

Fig. 7 is a flowchart describing a method for unpacking a simple object and is generally designated by the numeral 100. A simple head value of the packed simple object and a simple exit value of the packed simple object are retrieved, as indicated at 102. The simple head value is the starting point of the simple object in the transmitted binary representation. The simple exit value is the ending point of the simple object in the binary representation. Next a section of unpacked memory is allocated to accommodate the simple object, as indicated at 104. Finally, from the packed binary representation, the simple object size, type and value are copied into the unpacked memory location, as indicated at 106.

Fig. 8 is a flowchart describing a method for unpacking a complex object and is generally designated by the numeral 110. A complex head value of the packed complex object and a complex exit value of the packed complex object are retrieved, as indicated at 112. The complex head value is the starting point of the complex object in the transmitted binary representation. The complex exit value is the ending point of the complex object in the binary representation. Next a section of unpacked memory is allocated to accommodate the complex object, as indicated at 114. Finally, from the packed binary representation, the complex object type, value and size are copied into the unpacked memory location, as indicated at 116.

Fig. 9 is a flowchart describing a method for unpacking a list object and is generally designated by the numeral 120. A list head value of the packed list object and a list exit value of the packed list object are retrieved, as indicated at 122. The list head value is the starting point of the list object in the transmitted binary representation. The list exit value is the ending point of the list object in the binary representation. Next a section of unpacked memory is allocated to

accommodate the list object, as indicated at 124. Finally, from the packed binary representation, the list object array is copied into the unpacked memory location, as indicated at 126.

Fig. 10 is a flowchart describing a method for extracting a data element  
5 form the tagged data and is generally designated by the numeral 130. Method 10 begins with determining the type of data element contained within the tagged data, as indicated as 132. By way of example, a command such as query may return the data type. Optionally, query (TD, position) would return a tagged data type associated with the given position. Next, the data element is written into the  
10 tagged data object, as indicated at 134. A data value may be extracted by extract (TD, position) in order to be written into the tagged data object.

Fig. 11 is a flowchart describing a method for presenting data within a  
computing environment of the type having an application program interface  
prescribed by a data conversion and a wire formatting specification and is  
15 generally designated by the numeral 140. Method 140 begin with creating a tagged data object, as indicated at 142. Optionally, the tagged data object may be a universal data container that is platform independent, hardware architecture independent and language independent. The tagged data object may provide broad access to the manipulation and aggregation of structured data and unstructured  
20 data. Next a data element is encapsulated into the tagged data object, as indicated at 144. The encapsulation may provide tagged data which may include a corresponding tag identifier and the data element. The tagged data is labeled data that is physically written into a specific form. The tagged data is then packed into a wire format for transmission by converting the tagged data into a binary  
25 representation of the tagged data, as indicated at 146. Then the tagged data is transmitted, as indicated at 148. The tagged data is unpacked, as indicated at 150. Finally the data is extracted from the tagged data, as indicated at 152.

While there has been described what are believed to be the exemplary embodiments of the present invention, those skilled in the art will recognize that

other and further changes and modifications may be made thereto without departing from the scope of the invention which is defined by the appended claims, and it is intended to claim all such changes and modifications as fall within the true scope of the invention.